The Thesis committee for William Lin Certifies that this is the approved version of the following thesis:

Kubernetes Provenance

APPROVED BY

SUPERVISING COMMITTEE:

Vijaychidambaram Velayudhan Pillai, Supervisor
Christopher J. Rossbach, Co-Supervisor

Kubernetes Provenance

by

William Lin

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Computer Science

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2020

Dedicated to my family.

Acknowledgments

Many people have guided and advised me on my journey through undergrad and Master's program at UT Austin.

In particular I would like to thank the systems faculty at UT Austin who introduced me to the wonderful world of systems research and encouraged me to pursue graduate school.

The time I spent has a TA for Alison Norman's operating systems course has been invaluable to both my systems fundementals and ability to explain compex topics. In addition, she has been an role model and great mentor to me during my time at UT Austin.

I would like to thank my advisor, Vijay Chidambaram, for his awesome Virtualization class that first got me hooked on systems. He also showed great patience and gave great advice as I haphazardly first dipped my toes into research as an undergraduate student.

Chris Rossbach's courses particularly inspired me with his amazing slides. He also has been a great mentor who always, without fail, gave me new perspectives on different problems and ideas I encountered.

Kubernetes Provenance

William Lin, M.S.COMP.SC
The University of Texas at Austin, 2020

Supervisor: Vijaychidambaram Velayudhan Pillai
Co-Supervisor: Christopher J. Rossbach

The field of machine learning (ML) has experienced a period of renaissance since the 2000s. First, exponential increase in computational power and improvements in hardware has finally allowed machine learning algorithms to process the same amount of data in minutes and hours rather than hundreds of years. Second, the model of cloud computing made large scale clusters inexpensive and available to anyone at the click of a button, allowing them to scale their algorithms without having to personally maintain hundreds or even thousands of machines. However, despite the huge rise in popularity of machine learning in both research and industry, the ML community is facing a crisis of being able to reproduce results. Although the existing machine learning frameworks all have the ability to re-execute the same piece of code saved by a researcher, the typical workflow could involve different frameworks and accesses to data on remote machines. These cross-framework workflows can not be replicated by a single frameworks provenance system, and often

contain customized scripts and processes that can further obscure the ability for future replication and repeatability.

I make the argument in this thesis that because of machine learning's need for scale and frequent training on large clusters, Kubernetes serves as a good common layer for the systems community to interpose a layer of provenance collection to aid the ML community in reproducing results that make use of multiple machines, frameworks, and hardware platforms. In addition, I also propose two new mechanisms for collecting fine-grained provenance information from Kubernetes without modifying the application or host operating system.

Table of Contents

Acknowledgments **v**

Abstract **vi**

Chapter 1. Introduction **1**
1.1 Goals 2

Chapter 2. Background **4**
2.1 Provenance 4
2.2 Operating System Provenance 6
2.2.1 Application-level Provenance 7
2.3 Data Centers 8
2.4 Virtual Machines 9
2.5 Containers 9
2.5.1 Application Storage in Containers 10

Chapter 3. Kubernetes **12**
3.1 Cloud Computing Model 12
3.2 Microservice Architecture and Kubernetes 12
3.2.1 Integration with Kubernetes 14
3.3 Kubernetes Design and Architecture 14
3.3.1 Declarative Configuration 15
3.3.2 Processes and Pods 16
3.3.3 Storage 19
3.3.4 Networking and Services 20
3.3.4.1 Cluster Networking 21
3.3.4.2 Services 22

Chapter 4. Machine Learning and Kubernetes **23**
4.1 Machine Learning and Reproducibility 23
4.1.1 Non-determinism in Neural Networks 25
4.1.1.1 Datasets and their Transformations 25
4.1.1.2 Source of Data 27
4.1.1.3 Initialization of Parameters 29
4.1.1.4 Software, Drivers, and Hardware 30
4.2 Machine Learning Frameworks 31

Chapter 5. Kubernetes Provenance **34**
5.1 Kubernetes Auditing 35
5.2 Fine-Grained File System Access 39
5.2.1 FUSE Layer 40
5.2.2 FUSE 41
5.2.3 Adding FUSE to Pods 42
5.2.3.1 Granularity of Information 42
5.3 Pod to Pod Communication 43
5.3.1 Capture Mechanism 44
5.3.2 Identifying Accesses to Storage 45
5.3.3 Maintaining Security 46
5.4 Construction of Provenance Graph 46
5.5 Conclusion 47

Index **48**

Bibliography **49**

Vita **59**

Chapter 1

Introduction

The computing industry has slowly transitioned its focus in the last 15 years from on-premise server clusters to large scale data centers. In order to efficiently manage the machines exposed by cloud providers, developers have turned to orchestration systems such as Kubernetes (K8s) [14]. Orchestration systems provide necessary features for large scale cluster management such as auto-scaling, recovery and tolerance from failures, and ease of deployment. Benefitting from the large scale provided cheaply by the cloud providers, machine learning methods have experienced a renaissance period and become almost synonymous with data analytics. However, despite its popularity and widespread adoption, machine learning processes are plagued with the issue of reproducibility [40]. Failure to replicate existing research's results hampers new research at best. At worst, it could lead researchers to the wrong conclusions [20]. It is both in the interest of the machine learning community and the users of the machine learning models to have reproducible models, processes, and training workflows.

I believe the reproducibility problem outlined cannot be solved without the systems community examining the orchestration platforms on which these

machine learning models are so often trained on. While existing literature on provenance [47, 48, 51] has found interprocess communication difficult to capture on application level provenance systems, the same has also been true for cluster scale, distributed applications, such as machine learning frameworks. Although existing frameworks collect their own provenance data in addition to the algorithm source code, they still are unable to reproduce workflows and models that span across different machines, frameworks, or storage mediums [40]. To support provenance features that span across these boundaries normally isolated by the operating system, we need to leverage the information made available by the underlying orchestration systems running these frameworks.

1.1 Goals

The goals of this thesis are as follows:

1. Provide an overview of provenance literature, especially in the context of collecting provenance at the operating system level.

2. Provide an overview of the architecture and features of Kubernetes (K8s), the most popular orchestration system, and compare it to commodity operating systems.

3. Explore how existing machine learning frameworks support reproducibility.

4. Detail the challenges of providing provenance features for machine learning workloads in Kubernetes, incorporating insights from 1, 2, and 3.

5. Explore ideas and proposals for how orchestration systems and cloud operating systems can be leveraged to provide provenance features for machine learning tasks.

Chapter 2

Background

2.1 Provenance

Provenance, within computer science, means the lineage of data, and how processes and actors act on data. There exists a long line of research that explores both the application and the collection of provenance data in contexts ranging from databases to operating systems. Provenance data is immensely useful in answering questions related to "what happened?" in a system. For example, if a researcher is interested in reproducing a result stored in a particular file, they need know which process or binary created the file, if the file was ever written to after its creation and by whom, and for each of the processes that modified the file, what were the commands and inputs that created those processes. A provenance system designed with the ability to reproduce a file would not only need to efficiently collect and store data that can answer the questions listed, but also be able to quickly trigger the necessary actions and events in the system to automatically recreate the file. The key challenges when designing provenance systems are how to collect the minimum amount of data required and how to collect that information with minimal overhead. One can imagine an operating system that saves every single action performed on and by a process. It would be able to handle any

question thrown at it, but would take an impractical amount of storage and introduce crippling overhead.

The example above used provenance for reproducibility, but there are many other valuable usages for provenance:

1. *Retroactive security* is the detection of security and policy violations after execution by examining provenance data. It is often used to verify compliance with contractual or legal regulations, such as GDPR [10].

2. *Fault injection* is a technique where errors are intentionally introduced to test applications' resistance to failure. Provenance data and graphs can help identify key failure points in an application where injecting faults can have the greatest impact.

3. *Deletion and policy modification*: When handling sensitive data, it is important to know exactly where certain information is located, including within any derived files. During deletion or modification to access policies, it is important that every file containing the information is deleted or updated. A provenance system can achieve this by recording the relationship between objects and propagating operations to other objects derived from the original source.

4. *Anomaly detection*: For any environments where multiple executions or instances of an application are available, provenance systems can be used to record the normal behaviors of the application. Then, by creating a

model of normal behaviors, the executing task can be compared to detect abnormal behavior, signaling a fault or a security compromise.

Although in this thesis I specifically examine the application of provenance data for reproducibility of machine learning workflows, a provenance system for Kubenetes can be extended for other practical use cases.

2.2 Operating System Provenance

I focus on provenance systems at the operating systems level in particular, because orchestration systems are a form of distributed operating systems. There are distinct parallels between the design of commodity monolithic kernels and Kubernetes that allow us to apply the lessons and designs learned in existing works to Kubernetes. CamFlow[51] is a Linux Security Module (LSM) designed to capture data provenance for the purpose of system audit. CamFlow, as a LSM, utilizes the security hook system to cleanly integrate with Linux. LSM inserts two types of hooks or upcalls into the kernel. 1) When a kernel object (sockets, inode, files, etc.) is allocated and 2) when a kernel object is accessed. The two upcalls are used to create and maintain the necessary kernel data structures that store provenance data.

Some key challenges tackled by existing operating system level provenance systems include handling accesses to shared memory [48], interprocess communication [51, 52], and handling application level semantics [47]. Continuing the example from earlier, in order to reproduce a file, CamFlow would

need to keep track of accesses to every file in the system. This can quickly explode the amount of data we have to keep track of. CamFlow solves this by introducing two different modes of configuration: 1) whole provenance mode where all objects are captured and 2) selective mode that allows users to tailor the capture system to targets of interest. In selective mode, CamFlow allows a few different criterias by which users can select the capture target: pathname, network address, LSM security context, control group, and user and group ID.

2.2.1 Application-level Provenance

Muniswamy-Reddy et al.[47] showed that the application semantics need to be associated with provenance information from other layers of the system in order to effectively support provenance features. For example, if we want to keep track of files downloaded from the Web, simply recording the fact that the file was downloaded by the browser is not enough. Although most browsers can record the URL and the name of a downloaded file, if the file is moved or renamed, the link between the file and browser history is broken. Any attempts to associate the file back to the originate URL is impossible unless the provenance collection system also recorded information relevant to the specific application, which in this case is the browsers download history. The need for application aware provenance is particularly applicable in our case of exploring reproducibility for machine learning.

2.3 Data Centers

Instead of managing their own set of servers and hardware components on-premise, most companies have migrated their structure and applications to a cloud provider such as Amazon's AWS [1], Microsoft's Azure [15], or Google's GCP [9]. The reason behind their explosive popularity is simple: economies of scale allow these cloud platforms to provide machines to users on demand at drastically lower costs than each individual user purchasing and maintaining their own cluster of physical servers [36]. The large number of physical machines in each data center, on the orders of tens of thousands, allow providers to drastically lower the cost of maintaining each unit of server. By being able to pick a suitable location for temperature, data centers can optimize for the outside temperature and the centralized cooling methods employed. By being able to negotiate a contract with local governments and utility companies, they can obtain electricity at lower wholesale prices. By ordering large amounts of servers and spare parts for replacement, they are able to obtain a lower price directly from the manufactures than smaller companies or users can obtain at retail prices. Additionally, by renting machines from a cloud provider, the client never has to worry about the logistics of replacing parts, and upgrading servers. All of these reasons and many more lead to the effect that as a whole, cloud providers can supply bare-metal machines, virtual machines, and even specialized hardware such as GPUs [3, 5] and FPGAs [2, 4] at a much lower cost.

2.4 Virtual Machines

In the traditional model of a computer, the operating system manages and facilitates the interfacing of user software with the underlying physical resources of a machine, such as RAM, storage, CPU, and the network. Virtual machines [59, 64] are an emulation of the same set of physical hardware, often with the aid of specialized hardware. Virtual machines allow for a physical machine to be used to run multiple independent operating systems, each executing on its own set of emulated hardware all running on the same physical machine. While this emulation layer introduces overheads, it allows for different users of the virtual machines to run their workloads simultaneously on a single physical server. This results in an increased utilization of the underlying hardware compared to the application and its host operating system occupying the entire machine in the traditional model, yet not utilizing all of its resources.

2.5 Containers

At a high level, containers [45] are simply another way to package programs, similar to the ELF binary format for Linux. In addition to the binary, containers also include dependencies needed to execute the program contained. When executing a container, stricter restrictions [41] such as CPU utilization, memory usage, and network usage can be applied to the processes within the container. As a whole, containers allow the operating system to both ensure stronger isolation than processes, and also eliminate the need to maintain and

install dependencies and libraries.

An application's dependencies and environment are all packaged into the container image along with the process binary. As long as a container runtime exists on the machine, the application within the container can execute in an identical environment. Being able to maintain the same environment across different machines is a very powerful property; it allows developers to develop, test, and debug applications in the same runtime environment as the production machine.

Containers combined with virtual machines form a powerful tool that allows developers to quickly provision nodes and deploy their applications in a scalable manner without changing the runtime environment.

2.5.1 Application Storage in Containers

Container images are stored in a copy-on-write file format for disk images that is organized into multiple layers. When creating a new image, a developer can choose a base image, such as a bare Linux image, upon which to build the rest of the image. Then, more application specific libraries and binaries can be added on top as newer layers. These layers all become read-only during the execution of the container and thus is what allows the container image to act as the immutable source just as the ELF binary does for conventional Linux processes.

By default all files created and modified inside a container are stored on a writable layer. Unless part of the host file system is mounted into the con-

tainer at startup time, data will not persist when that container no longer exists. Thus, workloads that require writing to persistent storage need more coordination and management than simply moving the container image around, as the data stored on the host file system will remain on the same node. As we will discuss in the provenance chapter, this becomes particularly tricky when collecting provenance information for reproducibility.

Chapter 3

Kubernetes

3.1 Cloud Computing Model

Data centers, virtual machines, and containers all form the basis from which the cloud computing model has exploded in the industry. In order to obtain the maximum value from their physical machines, cloud providers offer most of their services through the form of virtual machines. Cloud providers supply virtual machine instances running on data center clusters to users who most commonly use containers to deploy and scale their applications on the cluster. This model also requires that the application be structured differently than the large monolithic application architecture.

3.2 Microservice Architecture and Kubernetes

Microservice is a pattern of organizing applications as loosely coupled units. This is in contrast to the traditional monolithic application with a single binary performing the functionality of the entire application. In a microservice architecture, each unit, known as a service, is designed to be highly specialized and communicate with other services using a well defined protocol. The protocol often takes the form of REST API or RPC, and has the benefit of

separating the interface from the underlying implementation. With the rise of cloud computing, applications that need to scale are required to replicate across different machines, and handling failures and recovering from them has become essential in maintaining high availability for the application.

Done correctly, applications built using the microservice architecture have the key advantages of being highly modular and scalable. Modular in the sense that other than the shared interface, the team responsible for a particular service does not need to know or care about the implementation details of any other services. This makes the development cycle of the service much shorter and easier than a huge, complex monolithic design.

The microservice architecture also boasts the benefit of better scalability. This is due to the fact that each service is implemented and deployed independently, allowing administrators to scale the number of replicas for services depending on how much load is put on a particular service in the application. On the other hand, a monolithic application has to be scaled as a unit by design, potentially wasting valuable compute, storage, and network resources on a less popular sector of the application.

As a result of growing popularity of microservices, software ecosystems have grown to support their management known as orchestration systems. Some of the most well-known are Kubernetes, Docker Swarm [8], and Nomad [18]. In this thesis we focus on the Kubernetes (K8s) framework which evolved out of the Borg [63] project at Google and is now open source. We chose this framework because of its popularity but the ideas and concepts we explore are

applicable to any other orchestration system.

3.2.1 Integration with Kubernetes

Every major cloud provider already has integration with Kubernetes. These integrations are only to the extent of deploying Kubernetes on a cluster. Management and configuration are still up to the developer. When deploying on a platform that does not have integration, such as a bare-metal cluster, there exist tools such as Kubeadm and Kubespray which perform the role of installing dependencies, setting up a default network configuration, and starting up all the necessary system pods.

3.3 Kubernetes Design and Architecture

Kubernetes is a platform that facilitates the automation and configuration of containerized workloads and services. In a production environment, especially for microservice applications, a large number of running containers and their replicas need to be configured and monitored. Orchestration platforms like Kubernetes provide a framework to efficiently run and monitor these distributed applications reliably. More specifically, Kubernetes provides the following functionalities:

- Service discovery and load balancing
- Storage orchestration
- Automated rollouts and rollbacks

- Automatic bin packing
- Self-healing
- Secrete and configuration management

The features supported by Kubernetes all build upon some basic components in the Kubernetes system and are managed by a control plane. In the next few sections, we introduce and describe each of the core components and discuss what information is required to efficiently provide provenance features in Kubernetes.

3.3.1 Declarative Configuration

The core design philosophy behind Kubernetes's control plane is that there exists the **desired state** of the system, specified by some entity such as an administrator and the **current state** of the system, supplied by each component to the control plane. The control plane of Kubernetes actively manages and configures each object so that their current state moves closer to the desired state until the two match.

Every object in Kubernetes includes two fields representing the desired and current state: 1) the object spec and 2) the object status, respectively. The spec field is defined before the creation of the object and is used to provide the initial description and configuration. The status field holds information regarding the current state of the object and is consumed and supplied by various systems of Kubernetes.

When creating an object in Kubernetes, the object spec must be provided that describes its desired state. The object spec is given to the API in JSON form, however it is usually specified by administrators in a YAML file and converted by the kubectl, a command line tool. Figure 3.1 shows an example of this YAML file for a Pod, which is described in the next section.

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: hello
spec:
  template:
    # This is the pod template
    spec:
      containers:
      - name: hello
        image: busybox
        command: ['sh', '-c', 'echo "Hello, Kubernetes!" && sleep 3600']
      restartPolicy: OnFailure
    # The pod template ends here
```

Figure 3.1: An example of a YAML file containing the spec for a Pod. Notably, the exact image used and command executed at container startup is also detailed in the spec.

3.3.2 Processes and Pods

The basic unit of scheduling and task encapsulation in Kubernetes is called a **Pod**. A Pod is analogous to a process in a commodity operating system such as Linux. Each Pod consists of a set of containers grouped together usually belonging to a single service of an application. Each replica of a Pod is scheduled by K8s to a particular machine that meets the optional resource

requirements specified by the Pods spec. Multiple Pods can be scheduled onto the same node (VM or bare-metal) and are all monitored and configured by a **kubelet**, a node agent daemon running on each node in the cluster.

The abstraction of **Pods** in Kubernetes replaced the more restrictive concept of **Jobs** in the Borg system [63]. A Job, in the Borg system, only encapsulates a single container running a set of identical programs and are too restrictive as the only grouping mechanism present in Borg. The lack of an abstraction for multi-job service as a single entity resulted in a disconnect between the level of abstraction provided by Borg, and the level users desired.

We now introduce a hypothetical website *appK8s* to illustrate the powerful abstraction provided by Pods. *appK8s* is broken down into two Pods A and B. Pod A contains two containers making up the front end website: 1) a container running Apache server serving web pages to the end user and 2) a container running a SQL database accessed by the container 1 to serve user data. Pod B on the other hand is responsible for managing user accounts and consists of a container running a service to authenticate user accounts, and a container for the database.

Because there are two different Pods, they can be scheduled and managed with different criterias and requirements. Number of replicas, memory requirement, memory limit, and cpu requirement can all be optionally specified in a Pods spec at creation time. Figure 3.2 shows an example of spec containing resource limits. For our *appK8s*, Pod A and Pod B can each have different amounts of replicas running if the demand for the services they pro-

vide are different. For instance if the number of user accounts are not high, but the website receives a huge amount of traffic.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: frontend
spec:
  containers:
  - name: db
    image: mysql
    env:
    - name: MYSQL_ROOT_PASSWORD
      value: "password"
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
  - name: wp
    image: wordpress
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
```

Figure 3.2: An example of source limits specified in a Pod's spec.

3.3.3 Storage

Within Kubernetes, the PersistentVolume subsystem abstracts the details of how storage is provided from how it is consumed. Kubernetes splits the management of storage into two API resources: PersistentVolume and PersistentVolumeClaim.

A **PersistentVolume** (PV) is a piece of storage that has been provisioned and made available in the cluster. A PVC can be either manually created by an administrator or automatically provisioned from a storage medium by Kubernetes. In either case, once created, a PVC becomes a resource in the cluster and can be used by a Pod for storage. The backing storage of the PVC is captured by the object spec and can be from NFS, iSCSI, or a cloud provider specific storage system, such as AWS's S3.

A **PersistentVolumeClaim** (PVC) is a request for storage by a user, and can specify resource limits and requirements similar to a Pod. Once created, a PVC can then be referenced by a Pod's spec for consumption in the containers within the pod.

Upon their creation Kubernetes will attempt to match the PVC to an available PV that meets the storage requirements defined in the PVC's spec. The creation of the Pod will also be blocked until all the PVCs referenced in the Pod's spec can be satisfied. Going back to our *appK8s* website, Pod A would require some backing storage for the SQL database running in the container. A PersistentVolume would first have to be created, then a PVC, then finally,

Pod A's spec would have to reference the PVC to signal to kubernetes that it requires some amount of storage to execute correctly. Figure 3.3 shows an example of a Pods spec referencing a PVC.

```
apiVersion: v1
kind: Pod
metadata:
  name: task-pv-pod
spec:
  volumes:
    - name: task-pv-storage
      persistentVolumeClaim:
        claimName: task-pv-claim
  containers:
    - name: task-pv-container
      image: nginx
      ports:
        - containerPort: 80
          name: "http-server"
      volumeMounts:
        - mountPath: "/usr/share/nginx/html"
          name: task-pv-storage
```

Figure 3.3: An example of a PersistentVolumeClaim being reference from a Pod's spec.

3.3.4 Networking and Services

In this section we give an overview of Kubernetes's network model and how users and administrators can manage and configure networks within the cluster. Networks within Kubernetes can be split into 4 distinct categories:

1. Container to container communications occur only within a single Pod. This is solved by simply using localhost communications within the Pod.

2. Pod to Pod communications are configured by Kubernetes. Every Pod is assigned an IP address automatically and is treated as a first-class citizen in the network model. Meaning that Pods can be viewed as a VM or physical host when considering tasks such as port allocation, load balancing, service discovery, configuration, and naming.

3. Pod to Service to External: A Service resource in Kubernetes is the abstraction around how an application running on a set of Pods is exposed to the external world as a network service.

3.3.4.1 Cluster Networking

Kubernetes employs what is known as the "IP-per-pod" model. Each IP address exists at the Pod scope and is shared by all the containers within one Pod. This means that only containers within the same Pod must coordinate port number usage. This model was chosen in part due to the lessons learned from the Borg system. Although Borg also isolated machine resources using containers, one IP address was assigned to every container running on a machine, causing difficulties with port isolation between Jobs on the same machine. The "IP-per-pod" also has the benefit of enabling low-effort porting of virtual machine applications to containers. Applications running on a virtual machine all share the same IP address and thus share the same port allocation and communication properties as a Kubernetes Pod.

Every Kubernetes cluster uses either proprietary or open source software-defined networks to establish the Pod to Pod network. A list of popular implementations of the Kubernetes networking model can be found in the K8s documentation [14]. Kubernetes only impose the following requirements on any networking implementation:

- Pods on a node can communicate with all Pods on all nodes without network address translation (NAT).
- Agents on a node (e.g. system daemons, kubelet) can communicate with all Pods on that node.

3.3.4.2 Services

Although the Pod's spec details the desired state of the object, attributes assigned to a pod at creation such as IP address, names, etc, are not guaranteed to be reused if the Pod dies. This leads to a problem of service discovery. For example, Pod B in *appk8s* provides the management of user accounts to the rest of the website; how can Pod A find and keep track of which IP addresses Pod B is assigned to. Or if some instances of Pod B crashes, what is the IP address Kubernetes assigned to the replacement Pods?

In Kubernetes, the **Service** resource provides the abstraction for a set of Pods exposing a service. A service can select a set of Pods based on labels and ports which allows for the underlying set of Pods to change without affecting the availability of the service.

Chapter 4

Machine Learning and Kubernetes

In this chapter we first provide an overview of the sources of non-determinism in machine learning, the root cause of the reproducibility crisis. Then, to demonstrate why the current provenance mechanism in existing machine learning frameworks are inadequate, we introduce three storage access patterns of Kubernetes machine learning frameworks and each pattern's impact on complete provenance collection.

4.1 Machine Learning and Reproducibility

There exist many types of machine learning methods each with their own strengths and weaknesses. Deep learning, or neural networks, in particular has become widely popular due to its ability to model and reproduce non-linear processes with high accuracy. Although techniques such as backpropagation used in neural networks has been known since the 1980s [56], it was not until the 2010 to 2012 when the exponential increasing in GPU computational power and new techniques [28] utilizing GPUs in deep learning algorithms converged, that deep learning took off on its meteoric rise in popularity. More recently, the availability of on-demand computational resources from cloud providers

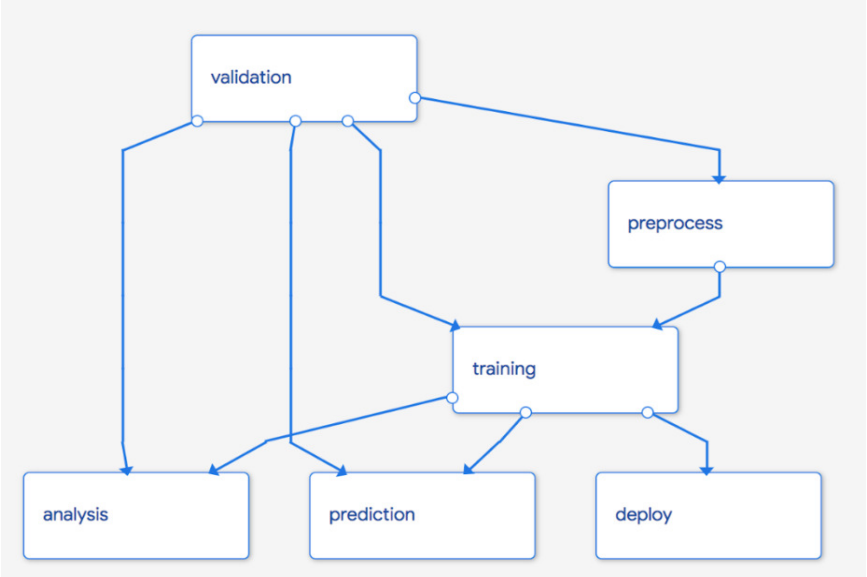

Figure 4.1: Each vertice represents an operation in the pipeline, and the edges represent the flow of data between each operation. Until all operations at the source vertices have completed, no new operations can begin execution.

has allowed users to train learning models at scale very cheaply, leading to wide spread usage of neural networks in many domains.

We focus on neural network models in this section due to: 1) its wide applications in areas such as image recognition [29], natural language processing [50], general game playing [58], finance [30], and medical diagnosis [26, 31] and 2) its position as the dominant type of machine learning workload in the cloud environment. However, despite its popularity, the machine learning community is facing a reproducibility crisis [40]. Randomness is a fundamental characteristic of neural networks and introduces numerous sources of non-determinism that causes results to vary even among "identical" runs.

4.1.1 Non-determinism in Neural Networks

Many tools and frameworks [21, 23] used for neural network training relies on the use of user code to specify the transformations and operations performed on data. However, even if the source code is available, it does not mean reproducibility is guaranteed. Oftentimes a component of the training process itself is randomized, or changes in the underlying computation resources introduce some perturbation that can affect the results. We now give a description of each source of randomness present in the process of training neural networks.

4.1.1.1 Datasets and their Transformations

Datasets are often shuffled randomly during various stages of training, introducing randomness that affects the overall reproducibility of the process. Consider a dataset used to train a neural network. As a preprocessing step, before training, the original dataset needs to be split into a training dataset, a validation dataset, and a test dataset. The need for these three separate datasets is important for preventing the model from overfitting to the particular dataset. **Overfitting** is when a model produced corresponds too closely, or even exactly, to the particular data. If overfitting occurs, the model may perform fantastically for the data used for training, but may fail to accurately predict future behavior.

After the split, the model is initially trained using the training set, and the validation dataset is used to provide an unbiased evaluation of the

model's performance during each iteration of the algorithm. Then, the test dataset is used to evaluate the performance of the model's prediction after training. It is essential that the three datasets are disjoint to prevent the machine learning algorithm from fitting too closely to the exact data points present in the training dataset.

A number of transformations are often performed on the data prior and during the training process that introduces randomness. We now give a list of examples that fit into this category:

- Before partitioning the initial dataset into the three pieces, the dataset itself often is randomly shuffled at initialization time or during a preprocessing step. An obvious case where this is needed is when the dataset is sorted according to some characteristic. In this case shuffling would ensure that the training, validation, and test datasets are representative of the overall distribution of the data.

- In a distributed training environment, such as Kubernetes machine learning frameworks, how the dataset is partitioned between machines could vary due to machine failure or changes in the resource constraints of a machine. Then, the variations could cause the resulting model to change despite the algorithm and the dataset remaining the same.

- Assuming that the method of splitting the dataset is deterministic, other steps can also introduce randomness. Many techniques and algorithms used in training neural network models also employ the shuffling of data

in between each iteration of the algorithm to prevent the model from learning unwanted correlations between data points. An example is batch gradient descent [42], a popular optimization where instead of using the entire training dataset to calculate the gradient for the current iteration, a subset of the dataset is used each iteration to save computation. In this variant, the training dataset is periodically reshuffled to ensure that the algorithm is not, by random chance, "stuck" with too many subsets that are unrepresentative.

Note that the list given above is not exhaustive nor are the bullet points mutually exclusive. These are examples of simple data manipulation that can occur before and during training that introduce sources of randomness, hampering reproducibility without additional mechanisms of provenance collection.

4.1.1.2 Source of Data

The datasets used in the training of neural network models are collected using a variety of methods. Often the data collection process is a continuous process as part of an application or service. As the training process is happening, more data points could be incorporated into the model, a technique known as online machine learning [24], or the new data points could simply be added to the initial dataset, from which the training, validation, and testing datasets are derived from. In both cases, without careful record keeping of the association between a model and the dataset used to train it, the newly

added data points are introducing unexpected sources of randomness into any existing reproducibility mechanisms.

For many machine learning problems, there exist a problem known as **class imbalance**, where the total number of a class of data is far larger than another class of data in a given real-world dataset. The reason why this is a particularly sinister problem for machine learning methods is that for most machine learning methods, including neural networks, function best when the number of instances of each class are roughly equal. To illustrate this we give an example below.

Given a dataset of financial transaction data, we would like to create a model to distinguish between genuine and fraudulent transactions. As the bank, we want to find as much fraudulent transactions as possible, since it is costly both for us and our customers. However, if this dataset of transactions consist of 10,000 genuine (negative) and 10 fraudulent (positive) transactions, any classifiers trained from this dataset will tend to classify fraudulent transactions as genuine ones (false negatives). The reason behind this is due to a conflict between the goal of machine learning methods and the goal of the bank creating this model. The goal of the training process is to minimize the loss function, which, in this case, will be the total number of mistakes made by the model when classifying transactions. While the bank wants to minimize false negatives. Taking this into account, the model will favor producing more false negatives than false positives since the dataset contains a much higher percentage of negatives (genuine transactions) than positives (fraudu-

lent transactions).

There are a number of solutions to the class imbalance problem, including modifying the loss function to take true positive rate and true negative rate into account. More recent approaches to this problem have involved an sampling based algorithm called Synthetic Minority Over-Sampling Technique (SMOTE) [27]. Where new data points belonging to the minority class are generated from its neighbors a random multiplier. Often done in a reprocessing operation, data augmentation [65] approaches can introduce a significant source of non-determinism if generated data are not saved as provenance information.

4.1.1.3 Initialization of Parameters

The choice of hyperparameters, or parameters chosen before the training process begins, can also have drastic effects on the quality of the result. Often these parameters are chosen manually by researchers and can undergo many versions before a good model is produced. If these non deterministic changes are not carefully recorded, it will be difficult to reproduce a particular model with only the source code.

For some hyperparameters the number of variables are both small, and their significance clear enough such that researchers and developers can use the source code to document their changes. Examples include *learning rate*, the step size at each iteration while moving toward a minimum of a loss function, and *mini-batch size*, a parameter used by batch gradient descent [42].

For other parameters, such as the initial weights of each layer in a neural network, they are set by sampling from a particular statistical distribution. Although the random initialization of layer weights has been shown to increase the speed of convergence over initializing all values to zero [35, 39], the non-determinism introduced is both significant and also hard to reproduce.

4.1.1.4 Software, Drivers, and Hardware

Updates to machine learning libraries can lead to subtle changes in behavior across different versions. Usually this is due to a change in the underlying implementation of a particular function in the library. Although less frequent, migrating a model from one framework to another may cause even larger discrepancies in the final performance of the model. For example, TensorFlow [22] explicitly warns users to not rely on consistent floating point values and random numbers across both minor and major version changes. Also, certain functions on cuDNN [6], the Nvidia Deep Neural Network library for GPUs, do not guarantee reproducibility across different runs. Consumers of cuDNN would need to avoid using these functions if they desire full reproducibility.

Without the support of Kubernetes's abstraction, machine learning frameworks simply do not have the privilege to access detailed system information needed to maintain consistency and collect provenance information regarding versioning information of hardware drivers and software dependencies.

Given that these challenges are common to all machine learning processes, whether distributed or local, how does Kubernetes and machine learning frameworks running on K8s give us hope for achieving provenance features?

4.2 Machine Learning Frameworks

Kubernetes machine learning frameworks are growing rapidly in popularity in part due to the abstractions provided by Kubernetes that facilitate efficient scaling of machine learning workloads. Although each of them provide mechanisms for reproducing a workflow, staying within a single layer of provenance collection is not enough to effectively maintain all the necessary provenance information across frameworks.

Machine learning frameworks at their core are a set of primitives used to specify data flow in a machine learning pipeline. A machine learning pipeline is a directed graph where the nodes represent operations performed on the training data such as transformations and actual training algorithms. The edges in the graph show how data flows from one operation to another. Until all the operations preceding a node is complete, and data has appeared on all the incoming edges, the node cannot begin its own operation. Figure 4.1 shows an example of this.

How operations are specified at each node is dependent upon each framework. Some frameworks supply either their own set of libraries and algorithms, while other frameworks simply coordinate the management and scheduling of data flow graphs generated by a separate, drop in, machine

learning library, such as Tensorflow [23].

We found that most machine learning frameworks use Kubernetes's storage systems in a few recurring patterns. We categorize these frameworks based on their storage patterns because of the large role that storage accesses play in provenance systems. Also, the management of storage has traditionally been a key difficulty when using containerized workloads. Thus we begin by examining how machine learning frameworks access storage.

We list the three ways a Pod can access storage and characterize their implications for a provenance capturing system:

- *Native PV*: is the idiomatic way to access storage in Kubernetes. This method uses a PVC and PV to specify and claim a storage allocation during the creation of the Pod. Since a Pod's spec contains the metadata associated with each PVC attached to Pod, necessary provenance data can be simply parsed from the Pod's spec.

- *In-cluster Service*: In this pattern, storage is exposed as a service on a separate Pod within the cluster. Other Pods access storage by making REST API requests or RPC calls. Although the storage service is known by other services within the cluster, Kubernetes inherently does not know the semantics of the interaction between Pods. To collect provenance for frameworks using this pattern, a mechanism to intercept Pod to Pod communication is necessary.

- *External service*: is when a Pod uses an external service provider to access storage. For example, instead of using a database within the cluster, a Pod could directly use a cloud provider's storage solution as the main backing store for data. We assume that frameworks do not access storage in this manner for our purpose of provenance collection for reproducibility.

Chapter 5

Kubernetes Provenance

Now I apply insights from operating system provenance to give an overview of how Kubernetes is a better fit for supporting provenance features than traditional operating system. In this chapter I also propose two new mechanism that can enable Kubernetes to support reproducibility of machine learning workloads.

Although fundamentally Kubernetes is still an operating system, the important distinction between Kubernetes and commodity operating systems is that Kubernetes is designed to have information be as transparently available as possible for debugging purposes [63]. On the other hand, performance and the overhead of system-level operations has traditionally been one of the main concerns for the designers and users of single node operating systems [43, 54, 55]. Rather than providing transparency for system events (system calls, interrupts, context switches, etc.) occurring inside of the kernel, it is imperative for system events on a local operating system to finish as fast as possible, due to their frequency and overall unpredictability. Whereas for Kubernetes, meaningful system-level events (object creation, termination, update, etc.) occur during clearly defined conditions such as 1) user or admin-

istrator updated an existing object, resulting in a cascade of events due to K8s attempting to reach the new desired state, 2) A new object is being created, such as a Pod, 3) An object has completed its task and has marked itself for termination and garbage collection, or 4) A failure has occurred and replacement instances are created.

In summary, although I initially compared a Pod to a process in a local operating system, Pods do not make "system calls" down to Kubernetes to perform tasks the same way process make system calls down to the kernel. Thus, system events are easier to collect in Kubernetes and can be used to reproduce the runtime state of a cluster. However, process are still executing within the Pods and they make application specific system calls and trigger events not able to be captured by Kubernetes. In this chapter I first give a description of the Auditing system in Kubernetes which is used to capture system-level events. Then, I describe important provenance information not able to be captured by the Auditing system, such as application system calls. Finally, we propose two new mechanism that will allow the relevant fine-grained information to be captured at the level of Kubernetes without modifying the host operating system or application code.

5.1 Kubernetes Auditing

The Auditing system in Kubernetes is a native mechanism designed for the collection of events and actions generated by users, administrators, or components of the system. As mentioned, this mechanism is possible due to

Kubernetess declarative configuration designed for ease of debugging. However, the auditing system only gives us information about components and interactions exposed to the Kubernetes system such as resource (Pods, PVs, PVCs, etc.) creation, termination, and modification. Pod to pod communication and fine grained information such as file-level access in PersistentVolumes need to be collected using a new mechanism.

First, we need to examine what information can already be provided to us by the Auditing system. The Pod spec is a great source of provenance data. By definition, an objects spec contains all the information necessary to define its desired state. In combination with the encapsulation of libraries and dependencies provided by the container, it might appear that Kubernetes naturally gives us reproducibility with the Pod abstraction. In fact, the benefit of being able to reuse the desired state again is one of the biggest advantages of using containers and orchestration systems. However, the act of Pod recreation in Kubernetes is designed for the microservice and stateless architecture. Meaning a number of assumptions about the containerized workload are required if our provenance system is to only rely on the information from the Auditing system:

- *Immutable PersistentVolumes*: Although the exact PV mounted into a Pod can be obtained by examining the Pods spec, fine-grained accesses are still opaque. Orchestration systems cannot obtain any fine-grained information from within the Pod, such as which files were changed. Other

than the initial creation and mounting of the PV into the Pod, the processes running inside containers also make file system calls to the underlying host OS rather than to Kubernetes. A simple solution around this problem would be to assume that each PV created is immutable after creation and can only be mounted read-only into subsequent consumer Pods of the data. This method implies that any changes made to a PV made by a Pod would need to be placed in a newly allocated PV. While not space efficient, it would be simple for a provenance system to track the exact input and output of any single instance of a Pod.

- *Containers are self-contained and deterministic*: If this is not the case, meaning containers change their behavior depending on some external input at runtime, then simply rerunning the Pod using the image named in the Pod's spec is not guaranteed to reproduce the exact results. Other factors not recorded in the Pods spec can modify and affect its behavior and extra provenance information would need to be collected to achieve true reproducibility. For example, consider a Pod that always requests data from another microservice during its execution. Now after its initial execution, we are attempting to reproduce its result by rerunning the exact same image. What happens if the microservice the Pod used is modified in between the two executions? What if the microservice has crashed? These are difficult questions to answer unless 1) we assume containers are always deterministic **or** 2) we can record the services the Pod in question communicated with, and can also replicate its behavior

during the rerun.

- *Pods communicate through PV storage*: Over the entire machine learning pipeline, data may need to be passed between Pods to satisfy the requirements of the data flow edges. However, Kubernetes can only capture that flow of data through the pipeline if the native PersistentVolume pattern is used. If another communication channel is used, such as through REST API or RPC, the communication would be lost to Kubernetes and the Auditing system.

- *No interprocess communication during container runtime* This assumption is needed for both determinism as well as the strict requirement of communication only though storage.

Any application or framework beyond a basic web server breaks these set of assumptions. In practice, Pod to Pod communication is at the core of the microservice architecture, and machine learning frameworks designed for K8s are no exception. How frameworks access storage also varies widely as seen in the three different patterns listed in the last chapter. In order for our provenance system to function without these restrictive assumptions, I now propose two different mechanisms for provenance collection that can provide us with the necessary information for reproducibility.

5.2 Fine-Grained File System Access

One method of collecting file access in Kubernetes would be to integrate every node in the cluster with an operating system provenance tool. For instance, CamFlow [51] collects fine-grained file system accesses using the upcall mechanism in Linux Security module. Information from the Auditing system could be integrated with Camflow to provide the association of Kubernetes objects to fine-grained access information. For example associating each PVs mounted into a Pod with the files modified, created, and deleted by the processes within that Pod.

There are four challenges with this approach:

- The existing API provided by CamFlow for integration with applications was designed as a collection mechanism for higher-level application objects. CamFlow consumes these object and integrates them with provenance records collected at the operating system level to provide provenance features. For our use case however, new provenance features would be provided at the layer of Kubernetes, since an application is split among multiple Pods and reproducing a result would require the mechanisms of K8s. Thus, Kubernetes would be consuming the provenance objects created by CamFlow rather than CamFlow consuming objects created by Kubernetes, as expected by the current architecture of CamFlow. A solution to this issue would require modification of CamFlow to export instead of receiving application objects.

- CamFlow requires that a system level descriptor (e.g., a file descriptor) is available to the application exporting provenance objects. Since we are trying to associate a PVC and Pod to its fine grained access information, our integration would be between Kubernetes and CamFlow. Although the container and the application will be using file descriptors to access files, Kubernetes is not privy to the actual file descriptor used from the abstraction level of the Pod. Source code modification of the process would be needed to extract each file descriptor allocated in the container.

- Requiring that CamFlow be present on each node in the cluster would severely complicate the installation of the provenance system. Existing clusters would have to be redeployed. And developers using platform-as-a-service (PaaS) providers, would be unable to easily modify their execution environments to install new dependencies. Also, CamFlow is a Linux LSM and would not be available for other operating systems such as Windows. Ideally, the solution would only require the recreation of Pods in the cluster, a common operation in Kubernetes.

- Coordination between all the nodes in the cluster would be needed to ensure the uniqueness of each provenance record. CamFlow only ensures uniqueness of records on a single machine.

5.2.1 FUSE Layer

Instead of integration with an operating system provenance tool, I propose to collect fine-grained access information by directing all file system ac-

cesses through a pass-through FUSE layer. First I give a brief overview of FUSE, then I discuss how the FUSE layer can be interposed between the Pod and the underlying storage without modifying application code.

5.2.2 FUSE

File systems serve as a common interface for applications to access data and have traditionally been part of the monolithic kernel in commodity operating systems. There do exist microkernel designs that implement file systems in user space, and discussions of the tradeoffs and performance differences can be found in [37]. More recently, user-space file systems have rose in popularity due to reasons stemming from 1) its flexibility in providing specialized functionality [25, 62] and 2) user space code is easier to develop, port, and maintain, resulting in many companies relying on user-space implementations such as Google's GFS [33], IBM's GPFS [57] and LTFS [53], RedHat's GlusterFS [11], etc.

Filesystem in Userspace (FUSE) is the most widely used user-space file system framework [60] with at least 100 file systems based on the framework [61]. Since the FUSE framework maintains the same Unix file system interface, it allows non-privileged users to create, and deploy a customized file system without modifying the application. Although the user defined file system code is executed in user-space, the FUSE module also enables access to the kernel interface as well, allowing for a pass-through style file system where fine-grained access can be recorded and then the operation is passed through

to the originally intended system call.

5.2.3 Adding FUSE to Pods

Kubernetes has control over the life-cycle of a Pod, including the creation of a Pod. During the creation of a Pod, the storage medium as well as the mount location inside the container is known to Kubernetes through the Pods spec. We can make use of this information to interpose our FUSE file system containing the provenance collection mechanism. Normally, Kubernetes will mount the original root directory on the host OS directly into the mount directory inside the container. To support FUSE, we mount the FUSE file system into the mount directory in the container with the original root directory on the host serving as the root directory of the FUSE.

This pass-through FUSE mounted into the container will simply record requests made to the underlying file system backing the PV. This mechanism will allow the collection of file-level access information with no modification of the cluster and the application.

5.2.3.1 Granularity of Information

An important aspect of a performant provenance system is minimizing the amount of data collected and the overhead of the collection mechanism. For our purpose of reproducing behavior at the Pod level, detailed information such as the exact file offset accessed is not needed. We only need to associate each file in the PV with modifications, if any, made by a particular Pod. In

the FUSE implementation, we only need intercept the open system call to record which files were opened, and the permission flags used in the system call. These records can then be inserted into the Pod creation event generated by the Auditing system to form an association between each PVC with a list of files accessed and whether they were only read or if they were also written to.

The Auditing events can then be used to construct a provenance graph that accurately represents the data flow edges between Pods. Without file-level access information, a provenance graph can still be constructed. However, without file-level access information, in order to maintain correctness, an dependency edge would be needed between any two Pods sharing a PV even if they accessed independent files within the PV. Thus, upon the command to reproduce a result from a particular Pod, every Pod that ever mounted the same PVs used would need to be rerun, potentially performing wasted computation.

5.3 Pod to Pod Communication

Pod to Pod communication during runtime immediately implies some form of IPC either through RPC or a REST API. Since Kubernetes cannot track these Pod to Pod communications, a new provenance mechanism to intercept network communication is required. Unlike the collection of file-level information, the need for the tracking of inter-Pod communication is not simply an important optimization for performance and resource utilization.

A provenance system that is unable to track this communication will not reproduce correctly. And the assumption that all containers are self-contained and deterministic is too restrictive.

To capture network communication, a network proxy can be inserted into every Pod as a sidecar container. **Sidecar containers** in a Pod are "helper" containers in addition to the application container and serve some auxiliary function such as monitoring, network address translation, or running batch jobs. The proxy container would serve to intercept network packets destined for storage services and create provenance records, allowing the construction of a provenance graph incorporating dependencies due to network communication.

5.3.1 Capture Mechanism

To efficiently capture and interpret relevant communication between Pods, we can make use of existing features and infrastructures already available in Kubernetes. Istio [13] is a service mesh layer that acts as a control plane for all networking related tasks between services in a cluster. Instead of individually configuring each running container and Pod, an administrator can simply submit a new network policy to the control plane managed by Istio.

Istio layers on top of Kuberentes and also relies on the pattern of sidecar containers that act as a "middleman" for all traffic between the different components. The control plane and these sidecar containers in each Pod work in tandem to provide configuration, monitoring, and security (service layer

encryption).

To support provenance capture, the sidecar container would need to be modified to be able to 1) identify all network communication relevant to supporting reproducibility, 2) capture and produce records that can be subsequently used to construct the provenance graph, and 3) still provide all of its existing functionalities, in particular the secure encryption of traffic.

5.3.2 Identifying Accesses to Storage

Simply capturing all network connections made between Pods would be enough for a correct provenance graph. However, blindly capturing traffic would both be wasteful and introduce unnecessary edges in the graph, causing extra containers to be rerun when reproducing the workload. Note that other than the native PV storage access pattern listed in the last chapter, there are also the in-cluster service, and external service pattern. Accesses to in-cluster storage services is the key to efficiently capturing network provenance data for reproducibility. Data is a crucial component of machine learning training, and if network accesses to in-cluster storage services can be recorded, a provenance graph can be constructed that accurately captures the data flow edges of the machine learning pipeline.

All of the machine learning frameworks we examined using the in-cluster service pattern use object stores [46] for the storage of user data or configuration data. Amazon's S3 API is the de facto standard for the object storage world, with multiple open source implementations as well as pro-

prietary implementations including Minio [16], Openstack Swift[19], Digital Ocean Spaces [7], and IBM Bluemix [12]. By targeting support for the Amazon S3 protocol, we can efficiently identify and capture all inter-Pod communication related to object store.

5.3.3 Maintaining Security

A sidecar container that can examine every network request introduces clear security vulnerabilities. Often microservice interactions are encrypted using a TLS protocol, and Istio by default is designed to provide encryption and access control management for the entire service mesh. The mechanism to support provenance should not require that the administrator turn off encrypted network traffic. However the capture mechanism proposed can occur in the sidecar container before traffic is encrypted and forwarded to the destination Pod's sidecar container, where it is decrypted and sent to the application container via localhost network.

5.4 Construction of Provenance Graph

Finally, after the collection of provenance records, the provenance graph needs to be constructed and efficiently queried to provide reproducibility at low cost. Once provenance relationship has been established, we can represent the graph either using a graph database, such as Neo4j [17], or an existing provenance protocol for maintaining provenance records in object stores, such as the protocols proposed by Muniswamy-Reddy et al. [49].

5.5 Conclusion

In summary, Kubernetes, as an orchestration system for distributed applications, inherently collects events and actions normally unavailable in a traditional local operating system. A provenance system can then exploit the declarative configuration style of Kubernetes to quickly create basic dependency graphs. For example, since PVC associates Pods to PVs, Kubernetes exposes the relationship between process and storage in each Pods spec. In a traditional operating system, an additional mechanism would have been required to extract these types of relationships, such as in the case of CamFlow using LSM.

However despite taking full advantage of Kubernetes's object specs, different challenges still remain in fully supporting reproducibility for machine learning workloads. In particular, file-level accesses and inter-Pod communication are not captured by Kubernetes. However, both are crucial when building an efficient and complete provenance graph. I examined each of these challenges and proposed two new mechanism for extracting the necessary provenance information without modifying application code or the host OS.

Index

Abstract, vi
Acknowledgments, v
Adding Fuse to Pods, 42
Application Storage in Containers, 10
Application-level Provenance, 7

Background, 4
Bibliography, 57

Capture Mechanism, 44
Cloud Computing Model, 12
Cluster Networking, 21
commands
    environments
        figure, 16, 18, 20, 23
Conclusion, 47
Construction of Provenance Graph, 46
Containers, 9

Data Centers, 8
Datasets and their Transformations, 25
Declarative Configuration, 15
Dedication, iv

Fine-Grained File System Access, 39
FUSE Layer, 40
FUSE, 41

Goals, 2
Granularity of Information, 42

Identifying Access to Storage, 45
Initialization of Parameters, 29
Integration with Kubernetes, 14
Introduction, 1

Kubernetes Auditing, 35
Kubernetes Design and Architecture, 14
Kubernetes Provenance, 34
Kubernetes, 12

Machine Learning and Kubernetes, 23
Machine Learning and Reproducibility, 23
Machine Learning Frameworks, 31
Maintaining Security, 46
Microservice Architecture and Kubernetes, 12

Networking and Services, 20
Non-determinism in Neural Networks, 25

Operating System Provenance, 6

Pod to Pod Communication, 43
Processes and Pods, 16
Provenance, 4

Services, 22
Software, Drivers, and Hardware, 30
Source of Data, 27
Storage, 19

Virtual Machines, 9

Bibliography

[1] Amazon web services: https://aws.amazon.com/.

[2] Aws fpga: https://docs.aws.amazon.com/awsec2/latest/userguide/fpga-getting-started.html.

[3] Aws gpu: https://docs.aws.amazon.com/dlami/latest/devguide/gpu.html.

[4] Azure fpga: https://docs.microsoft.com/en-us/azure/machine-learning/how-to-deploy-fpga-web-service.

[5] Azure gpu: https://docs.microsoft.com/en-us/azure/virtual-machines/sizes-gpu.

[6] cudnn: https://developer.nvidia.com/cudnn.

[7] Digital ocean spaces: https://developers.digitalocean.com/documentation/spaces/.

[8] Docker swarm: https://docs.docker.com/get-started/swarm-deploy/.

[9] Gcp: https://cloud.google.com/.

[10] Gdpr: https://gdpr-info.eu/.

[11] Glusterfs: https://www.gluster.org/.

[12] Ibm bluemix: https://console.ng.bluemix.net/catalog/.

[13] Istio: https://istio.io/.

[14] Kubernetes: https://kubernetes.io/docs/home/.

[15] Microsoft azure: https://azure.microsoft.com/en-us/.

[16] Mino: https://min.io/.

[17] Neo4j: https://neo4j.com/.

[18] Nomad: https://www.nomadproject.io/.

[19] Openstack swift: https://wiki.openstack.org/wiki/swift.

[20] Reproducibility: https://www.nature.com/news/1-500-scientists-lift-the-lid-on-reproducibility-1.19970.

[21] Scikit-learn: https://scikit-learn.org/stable/index.html.

[22] Tensorflow: https://www.tensorflow.org/.

[23] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI16, page 265283, USA, 2016. USENIX Association.

[24] András A. Benczúr, Levente Kocsis, and Róbert Pálovics. Online machine learning in big data streams. *CoRR*, abs/1802.05872, 2018.

[25] John Bent, Garth Gibson, Gary Grider, Ben McClelland, Paul Nowoczynski, James Nunez, Milo Polte, and Meghan Wingate. Plfs: A checkpoint filesystem for parallel applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC 09, New York, NY, USA, 2009. Association for Computing Machinery.

[26] Leonardo Bottaci, Philip J. Drew, John E. Hartley, Matthew B. Hadfield, Ridzuan Farouk, Peter WR Lee, Iain MC Macintyre, Graeme S. Duthie, and John RT Monson. Artificial neural networks applied to outcome prediction for colorectal cancer patients in separate institutions. *The Lancet*, 350(9076):469–472, Aug 1997.

[27] Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. Smote: Synthetic minority over-sampling technique. *J. Artif. Int. Res.*, 16(1):321357, June 2002.

[28] Dan C. Cireşan, Ueli Meier, Jonathan Masci, Luca M. Gambardella, and Jürgen Schmidhuber. Flexible, high performance convolutional neural networks for image classification. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Volume Two*, IJCAI11, page 12371242. AAAI Press, 2011.

[29] Dan Cirean, Ueli Meier, Jonathan Masci, and Jrgen Schmidhuber. Multi-

column deep neural network for traffic sign classification. *Neural Networks*, 32:333 – 338, 2012. Selected Papers from IJCNN 2011.

[30] Jordan French. The time travellers capm. *Investment Analysts Journal*, 46(2):81–96, 2017.

[31] Dr. Ganesan, N., Dr. Venkatesh, K., Dr. Rama, M. A., and A. Malathi Palani. Application of Neural Networks in Diagnosing Cancer Disease using Demographic Data. *International Journal of Computer Applications*, 1(26):81–97, February 2010.

[32] Ashish Gehani and Dawood Tariq. Spade: Support for provenance auditing in distributed environments. In *Proceedings of the 13th International Middleware Conference*, Middleware 12, page 101120, Berlin, Heidelberg, 2012. Springer-Verlag.

[33] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP 03, page 2943, New York, NY, USA, 2003. Association for Computing Machinery.

[34] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):2943, October 2003.

[35] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In Yee Whye Teh and Mike Titterington, editors, *Proceedings of the Thirteenth International Conference*

on Artificial Intelligence and Statistics, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy, 13–15 May 2010. PMLR.

[36] Albert Greenberg, James Hamilton, David A. Maltz, and Parveen Patel. The cost of a cloud: Research problems in data center networks. *SIGCOMM Comput. Commun. Rev.*, 39(1):6873, December 2009.

[37] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of -kernel-based systems. *SIGOPS Oper. Syst. Rev.*, 31(5):6677, October 1997.

[38] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of -kernel-based systems. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP 97, page 6677, New York, NY, USA, 1997. Association for Computing Machinery.

[39] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *CoRR*, abs/1502.01852, 2015.

[40] Matthew Hutson. Artificial intelligence faces reproducibility crisis. *Science*, 359(6377):725–726, 2018.

[41] Poul-Henning Kamp and Robert NM Watson. Jails: Confining the omnipotent root. In *Proceedings of the 2nd International SANE Conference*,

volume 43, page 116, 2000.

[42] Mu Li, Tong Zhang, Yuqiang Chen, and Alexander J. Smola. Efficient mini-batch training for stochastic optimization. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD 14, page 661670, New York, NY, USA, 2014. Association for Computing Machinery.

[43] Jochen Liedtke. Improving ipc by kernel design. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP 93, page 175188, New York, NY, USA, 1993. Association for Computing Machinery.

[44] Jochen Liedtke. Improving ipc by kernel design. *SIGOPS Oper. Syst. Rev.*, 27(5):175188, December 1993.

[45] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2, 2014.

[46] M. Mesnier, G. R. Ganger, and E. Riedel. Object-based storage. *IEEE Communications Magazine*, 41(8):84–90, 2003.

[47] Kiran-Kumar Muniswamy-Reddy, Uri Braun, David A. Holland, Peter Macko, Diana Maclean, Daniel Margo, Margo Seltzer, and Robin Smogor. Layering in provenance systems. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, USENIX09, page 10, USA, 2009. USENIX Association.

[48] Kiran-Kumar Muniswamy-Reddy, David A. Holland, Uri Braun, and Margo Seltzer. Provenance-aware storage systems. In *Proceedings of the Annual Conference on USENIX 06 Annual Technical Conference*, ATEC 06, page 4, USA, 2006. USENIX Association.

[49] Kiran-Kumar Muniswamy-Reddy, Peter Macko, and Margo Seltzer. Provenance for the cloud. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, FAST10, page 1514, USA, 2010. USENIX Association.

[50] Daniel W. Otter, Julian R. Medina, and Jugal K. Kalita. A survey of the usages of deep learning in natural language processing. *CoRR*, abs/1807.10854, 2018.

[51] Thomas Pasquier, Xueyuan Han, Mark Goldstein, Thomas Moyer, David Eyers, Margo Seltzer, and Jean Bacon. Practical whole-system provenance capture. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC 17, page 405418, New York, NY, USA, 2017. Association for Computing Machinery.

[52] Thomas F. J.-M. Pasquier, Jatinder Singh, David M. Eyers, and Jean Bacon. Camflow: Managed data-sharing for cloud services. *CoRR*, abs/1506.04391, 2015.

[53] D. Pease, A. Amir, L. V. Real, B. Biskeborn, M. Richmond, and A. Abe. The linear tape file system. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–8, 2010.

[54] Xiang (Jenny) Ren, Kirk Rodrigues, Luyuan Chen, Camilo Vega, Michael Stumm, and Ding Yuan. An analysis of performance evolution of linuxs core operations. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP 19, page 554569, New York, NY, USA, 2019. Association for Computing Machinery.

[55] Yaoping Ruan and Vivek Pai. Making the "box" transparent: System call performance as a first-class result. pages 1–14, 01 2004.

[56] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, Oct 1986.

[57] Frank Schmuck and Roger Haskin. Gpfs: A shared-disk file system for large computing clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST02, page 16, USA, 2002. USENIX Association.

[58] David Silver, Aja Huang, Christopher J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–503, 2016.

[59] J. E. Smith and Ravi Nair. The architecture of virtual machines. *Computer*, 38(5):32–38, 2005.

[60] M. Szeredi. Filesystem in userspace: http://fuse.sourceforge.net/.

[61] Vasily Tarasov, Abhishek Gupta, Kumar Sourav, Sagar Trehan, and Erez Zadok. Terra incognita: On the practicality of user-space file systems. In *7th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 15)*, Santa Clara, CA, July 2015. USENIX Association.

[62] Cristian Ungureanu, Benjamin Atkin, Akshat Aranya, Salil Gokhale, Stephen Rago, Grzegorz Całkowski, Cezary Dubnicki, and Aniruddha Bohra. Hydrafs: A high-throughput file system for the hydrastor content-addressable storage system. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, FAST10, page 17, USA, 2010. USENIX Association.

[63] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015.

[64] Carl A. Waldspurger. Memory resource management in vmware esx server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181194, December 2003.

[65] Sebastien C. Wong, Adam Gatt, Victor Stamatescu, and Mark D. McDonnell. Understanding data augmentation for classification: when to

warp? *CoRR*, abs/1609.08764, 2016.

Vita

William Lin attended Allen High School in Allen, Texas. In 2015 he began his undergraduate studies in computer science at UT Austin.

Email address: wlsaidhi@gmail.com

This thesis was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.